Trabajo de Fin de Grado

UPC-ETSEIB Ingeniería Industrial

Puerta de enlace basada en Python para conectar un USBtin con el software BusMaster

MEMORIA

Autor: Melchor Sanz Sesé
Director: Manuel Moreno Eguílaz

Convocatoria: Junio 2018

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Resumen

Actualmente el bus CAN (Controller Area Network) es de vital importancia para la trasmisión de información entre sensores, actuadores y ECUs (Electronic Control Unit) en cualquier sistema gobernado electrónicamente. La aplicación más común la podemos encontrar en el sector del automóvil, donde diferentes sensores y actuadores del vehículo se conectan al bus y mediante varias ECUs se controla diferentes parámetros de conducción y seguridad (ABS, Airbag…).

En este proyecto se trabaja con el hardware USBtin, un conversor USB/CAN de muy bajo coste fabricado en Alemania, ideal para iniciarse en el mundo del bus CAN.

Existe un software de libre distribución denominado USBtinViewer, desarrollado por el mismo inventor del USBtin, que permite ver la información que envía y recibe el USBtin. El problema es que es demasiado sencillo en cuanto al procesamiento de datos, mostrando únicamente los mensajes de bus CAN en forma de texto. Este es el motivo por el cual se pretende comunicar el USBtin con la potente interfaz gráfica que ofrece BusMaster, un software “Open Source” muy utilizado por la comunidad internacional que se dedica a buses de comunicaciones en automoción. Esta interfaz usa un bus virtual que, a diferencia de USBtinViwer, es capaz de graficar los datos en tiempo real además de permitir trabajar con bases de datos de bus CAN. BusMaster trabaja con varios tipos de hardware de bus CAN, pero no con USBtin, por tanto, éste será el fundamento principal de este proyecto, hacer compatible el hardware USBtin con el software BusMaster.

En consecuencia, el objetivo principal del proyecto es crear una puerta de enlace entre el hardware de bajo coste USBtin y el software “Open Source” BusMaster. Esta puerta de enlace es un programa de ordenador escrito en lenguaje Python capaz de comunicarse tanto con el USBtin mediante una conexión serie USB como con el BusMaster mediante un bus virtual. De esta manera, se podrá controlar toda la información transmitida por el bus CAN y, además, con la ayuda que nos proporciona el software BusMaster, el procesamiento de datos transferidos por el USBtin podrá ser mucho más efectivo y útil que no utilizando el software por defecto que ofrece USBtin.

Sumario

1. Glosario

CAN: Controller Area Network

ECU: Electronic Control Unit

USB: Universal Serial Bus

GUI: Graphical User Interface (interfaz gráfica)

Python Shell: Ventana ejecución de Python

COM: Puerto serie del PC

2. Prefacio

2.1. Origen del proyecto

Durante los últimos años, en la sección Sur del Departamento de Ingeniería Electrónica de la UPC se han llevado a cabo varios trabajos (TFGs/TFMs) basados en la transferencia de datos mediante el uso de un bus CAN [1]. Más concretamente, se ha utilizado un dispositivo de muy bajo coste, capaz de enviar y recibir mensajes de bus CAN, denominado USBtin [2].

El origen de este proyecto viene dado por la necesidad de poder monitorizar de forma gráfica los datos transferidos por dicho dispositivo hardware (USBtin).

2.2. Motivación

Actualmente, el sector de la automoción depende, entre otros muchos factores, de la electrónica. Dentro de toda la electrónica de un coche, el bus CAN es de vital importancia para el correcto funcionamiento del vehículo ya que aporta infinidad de datos importantes como: velocidad, temperatura, presión, estado del vehículo, estado de los sensores, etc.

Por otra parte, adquirir conocimientos en un lenguaje de programación como Python y manejar diferentes tipos de bibliotecas y métodos como los que ofrece este lenguaje es muy interesante de cara al sector informático, tal ligado actualmente al electrónico.

2.3. Requerimientos previos

Para un correcto desarrollo del proyecto, se requieren conocimientos básicos de programación en Python, tales como:

- Dominar las herramientas básicas (*for, while, if…*)
- Saber como funciona una clase (*Class*) y saber utilizar métodos de otras clases guardadas en diferentes ficheros.
- Importar diferentes bibliotecas y utilizarlas correctamente

Es conveniente entender distintos sistemas de codificación, como, por ejemplo, el sistema binario, decimal y hexadecimal, entre otros.

2.4. Problemática a resolver y alternativas

Se dispone, por un lado, del hardware USBtin [2], un conversor CAN/USB de bajo coste. Por otro lado, existe el software “Open Source”, denominado BusMaster, muy flexible y versátil para monitorizar mensajes de bus CAN. El problema es que ambos son incompatibles, es decir, hasta la fecha dicho software no puede controlar el mencionado hardware, tal y como muestra la Fig. 2.4.1.

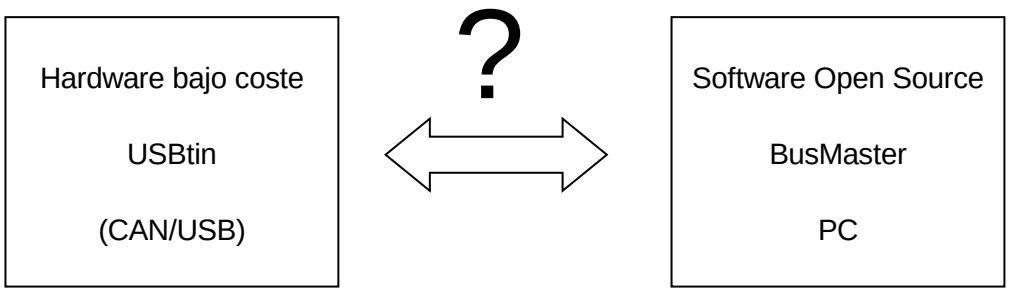


Fig. 2.4.1. Problemática que se quiere resolver. Fuente: propia.

Para hacer que ambos, es decir, USBtin y BusMaster sean compatibles, existen varias opciones:

- Desarrollar un “driver” para USBtin, de manera que el software BusMaster sea 100% compatible. Esta vía parece la más razonable. Sin embargo, el tiempo y los conocimientos necesarios para ello pueden llegar a ser demasiado grandes.

- Desarrollar una puerta de enlace que, de forma simultánea, se comunique con el software BusMaster y con el hardware USBtin, pasando los mensajes de uno hacia el otro y viceversa. Para ello es necesario algún sistema de comunicación con el BusMaster. La empresa Kvaser, de forma gratuita, ofrece el módulo canlib [12], que una vez instalado, permite crear un bus CAN virtual dentro del ordenador. El software BusMaster es compatible con el bus virtual de Kvaser y además, usando cualquier lenguaje de programación, se puede acceder de forma muy sencilla a dicho bus. En consecuencia, la alternativa escogida en este proyecto utiliza un bus virtual, tal y como se muestra en la Fig. 2.4.2.

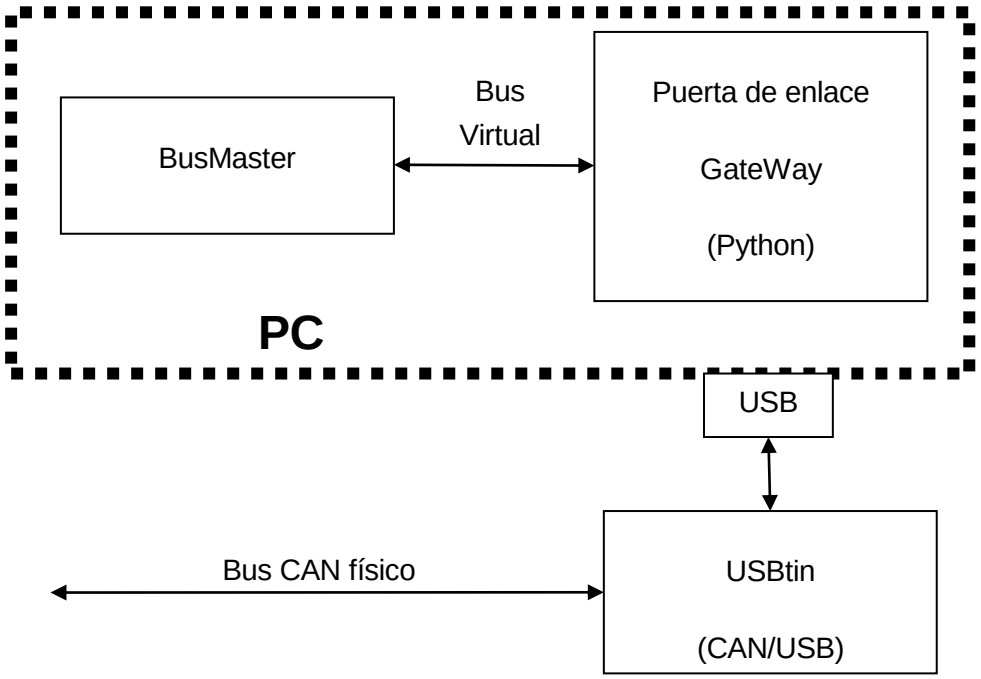


Fig. 2.4.2. Alternativa escogida basada en un bus virtual y una puerta de enlace.

3. Introducción

3.1. Objetivos del proyecto

El objetivo principal del proyecto es poder utilizar el software BusMaster para poder monitorizar de forma gráfica los datos que circulan por un bus CAN, que son capturados mediante el hardware USBtin. Los mensajes se pueden envíar tanto de BusMaster como desde el propio hardware USBtin.

Herramientas utilizadas

A parte de todos los componentes citados anteriormente, durante el proyecto, se utiliza otro software gratuito denominado, CanKing [3]. Para utilizar dicho software es necesario un hardware no gratuito ofrecido por Kvaser [7]. Se utiliza junto al USBtin, conectados entre ambos, de manera que desde el software libre ofrecido por Kvaser (CanKing), podemos programar los mensajes que queremos enviar al bus CAN.

Así pues, la idea principal es conseguir dominar las diferentes herramientas ofrecidas para todos los componentes que vamos a utilizar durante el desarrollo del proyecto:

A. Interfaz gráfica BusMaster.

B. Herramientas y comandos para utilizar USBtin.

C. Uso del CanKing junto con el hardware HS Leaf Light de Kvaser [7].

3.2. Alcance del proyecto

El alcance de este proyecto es crear la compatibilidad entre el USBtin y el software BusMaster para así, poder graficar los datos enviados por el bus CAN. Crear esta puerta de enlace supone aprender sobre el lenguaje de programación Python, el cual tiene varias bibliotecas y comandos específicos para poder manipular tanto BusMaster como el USBtin.

Así pues, este proyecto consigue ampliar el campo de trabajo del USBtin, haciéndolo compatible con el software “open source” denominado BusMaster. Cabe destacar que BusMaster es compatible con bases de datos de bus CAN, las cuales se pueden crear y modificar gracias al programa Kvaser Database Editor 3 [4], y así, poder decodificar la información que lleva incluida cada una de las tramas de bus CAN.

En resumen, este proyecto permite que BusMaster sea compatible con USBtin, pero para ello se necesitan algunas herramientas adicionales, como por ejemplo una base de datos de bus CAN para poder decodificar la información y así, poder ver un gráfico en tiempo real coherente en BusMaster. Es decir, de cara al objetivo de este proyecto, no se pueden graficar los mensajes de un bus CAN cualquiera, ya que cada uno va asociado a una base de datos específica y los resultados no serían decodificables. Sin embargo, si se conoce la procedencia y el significado exacto de los mensajes que circulan en el bus, la aplicación es directa y efectiva.

4. Solución

Antes de comenzar con la solución es necesario repasar qué programas y dispositivos se deben conocer para poder entender este documento sin dificultades. A continuación, se presentan acompañados de una breve descripción.

BusMaster v3.2.2 [5]

Software *open source* capaz de enviar y recibir mensajes de bus CAN, ya sea mediante un hardware compatible o mediante un bus virtual. Esta dotado con herramientas varias para poder procesar los datos transmitidos por el bus. Este programa es incapaz de trabajar directamente con el USBtin pero sí con dispositivos de la empresa Kvaser [6], por ejemplo.

Kvaser CanKing [3]

CanKing para Windows es un monitor de bus CAN y una herramienta de diagnóstico de propósito general. Es especialmente adecuado para el trabajo de desarrollo interactivo. Los mensajes CAN se pueden enviar fácilmente y se puede observar el impacto correspondiente en el módulo objetivo. No es compatible con USBtin, pero sí con, por ejemplo, el hardware HS Leaf Light de Kvaser [7].

USBtin

USBtin es un conversor CAN/USB de muy bajo coste, diseñado y fabricado por el ingeniero alemán T. Fischl [8].

Según T. Fischl [9], “es una interfaz simple de USB a CAN. Puede monitorizar buses CAN y transmitir mensajes CAN. USBtin implementa la clase USB CDC y crea un comportamiento virtual en el host del ordenador.”

Python

Python es un lenguaje de programación muy versátil. Destaca debido a su hincapié por utilizar un código legible. Es un lenguaje de programación multiparadigma, Según lo describe B. Stoustrup, [10] “Los lenguajes de programación multiparadigma permiten crear programas usando más de un estilo de programación”. Esto quiere decir que soporta programación orientada a objetos, programación imperativa y programación funcional.

La instalación estándar de Python incluye un módulo adicional denominado Tkinter, cuyas herramientas están diseñadas para crear interfaces gráficas o GUIs. Una de las partes importantes de la puerta de enlace es una pequeña interfaz gráfica programada gracias a esta extensión de Python. Esta GUI está directamente relacionada con la puerta de enlace *GatewayUSBtin.py.*

La solución al problema se realizará de una forma fragmentada paso a paso. Primero, se trabaja el fichero base de este proyecto (*GatewayUSBtin.py*). En él se encuentran las herramientas necesarias para conectarse al bus CAN real y poder leer, tanto mensajes enviados desde USBtin, como desde BusMaster. Dentro de este fichero se halla una clase con dos funciones principales, una encargada de leer mensajes enviados desde USBtin y la otra está atenta a los mensajes enviados desde BusMaster.

Por otro lado, se crea el fichero *GUI_GatewayUSBtin.py*. Éste contendrá la interfaz gráfica que nos proporcionará diferentes opciones de los principales parámetros que se deben tener en cuenta a la hora de conectarse al bus CAN. Así pues, la solución se divide en dos grandes bloques, el fichero principal y la interfaz gráfica.

4.1. GatewayUSBtin

En el siguiente apartado se explica el contenido del programa *Gateway_USBtin.py*. La función principal de este programa es servir de puerta de enlace entre el conversor USBtin y el software de ordenador BusMaster. Debido a la extensión del programa, se incluye el código completo en el Anexo 1, de manera que todas las referencias a dicho código son fácilmente accesibles. Para una correcta comprensión del programa, también se hace referencia a varios ficheros utilizados en el proyecto. Todos estos ficheros también se encuentran en el Anexo 1.

GatewayUSBtin se encarga principalmente de hacer compatibles el USBtin con BusMaster mediante un código escrito en lenguaje de programación Python. Este programa, mientras se está ejecutando, realiza dos funciones básicas: leer todos los mensajes que provienen del USBtin enviándolos a BusMaster y viceversa, es decir, leer todos los mensajes que provienen de BusMaster para enviarlos hacia USBtin. Para leer los mensajes que provienen del USBtin es necesario acceder al puerto serie COM mediante el módulo de Python pyserial [13]. Para leer los mensajes que provienen de BusMaster es necesario acceder al bus virtual creado por los drivers de bus CAN del fabricante Kvaser [14].

La puerta de enlace o Gateway desarrollada y encapsulada en una clase de Python realiza las siguientes funciones:

- Se encarga de configurar la velocidad (baudrate) del hardware USBtin.
- Se encarga de conectar el hardware USBtin al bus CAN (bus ON).
- Se encarga de configurar las máscaras y filtros del hardware USBtin.
- Se encarga de leer periódicamente del hardware USBtin los mensajes que viajan por el bus CAN real.
- Se encarga de enviar hacia el BusMaster mediante un bus virtual todos los mensajes recibidos por el USBtin.
- Se encarga de recibir los mensajes generados por el BusMaster y enviarlos al USBtin para que viajen por el bus CAN.

Para empezar, se explican las diferentes bibliotecas que se utilizan desde el programa *GatewayUSBtin.py.* Como podemos ver al principio del programa *(GatewayUSBtin – líneas 1-7)*, se importan una serie de extensiones propias de python y alguna biblioteca aparte,

más específica de USBtin. Las bibliotecas propias de python son: time, logging y threading, mientras que las bibliotecas más específicas y que se encuentran en el anexo son: usbTinLib, PeriodicThread y constantsUSB. También se importa una biblioteca específica de CAN, denominada canlib y que permite conectarse con BusMaster mediante un bus virtual.

El programa *GatewayUSBtin.py* consta principalmente de una clase donde se encuentran los principales métodos para conectarse tanto al bus CAN real como al bus virtual. Como en cada clase de Python, se define el *__init__ (GatewayUSBtin.py – líneas 11-19),* en él se instancia la clase USBtin, se arrancan dos hilos *(véase capi-4.3.2)*, y se abre un canal del bus virtual. Las propiedades principales que se pueden escoger a la hora de instancian la clase USBtin son: *"COMport", "Baudrate" y "modo"*. Estas propiedades se escogen antes de conectarte al bus mediante USBtin *(GatewayUSBtin.py – línea 11).*

A continuación del *__init__,* se encuentran los dos métodos principales de la clase, *LeerMensajesUSBtin() (GatewayUSBtin.py – líneas 21-41)* y *LeerMensajesBusMaster() (GatewayUSBtin.py – líneas 42-57).* Ambos métodos realizan la misma función pero de manera inversa, es decir, la primera lee constantemente mensajes enviados desde el conversor USBtin al bus CAN y, los devuelve de manera que pueda ser recibido por BusMaster. La segunda función trabaja inversamente, leyendo mensajes enviados desde BusMaster y los envía al USBtin.

Dentro de esta misma clase se hallan otras funciones como *setMask() (GatewayUSBtin.py – líneas 59-74) y setFilter() (GatewayUSBtin.py – líneas 75-91).* Estas funciones son para filtrar los mensajes que se quieren recibir. Más adelante se explica con más detalles el funcionamiento de las mascaras y filtros, pero, de forma resumida, según el valor de la mascara ('0' o '1') se activa el filtro para recibir mensajes con un identificador determinado. Ambas funciones requieren de un parámetro de entrada.

Seguidamente, se encuentran tres funciones que sirven para activar la puerta de enlace y detenerla. Una de esta funciones es *start() (GatewayUSBtin.py – líneas 92-99),* otra es *stop() (GatewayUSBtin.py – líneas 100-107)* y la última es *CloseAll() (GatewayUSBtin.py – líneas 107-113).* La primera de ellas abre un canal del virtual y los dos hilos, la segunda lo cierra y la tercera, aparte de cerrar lo mismo que la segunda, también cierra el puerto serie.

Este programa es el encargado de procesar los datos ente USBtin y BusMaster. El problema es la dificultad de poder cambiar las principales propiedades de conexión en el bus CAN (Puerto serie, Baudrate, Máscaras y Filtros). Por ello, se diseña un segundo programa totalmente ligado a *GatewayUSBtin.py* que proporcionará una pequeña interfaz gráfica desde la cual se podrán modificar los valores citados anteriormente, es decir, conectarse y desconectarse del bus CAN.

En el siguiente capítulo se explica el funcionamiento de la interfaz gráfica incluida en el fichero *GUI_GatewayUSBtin.py*, gracias al cual el uso de esta puerta de enlace será mucho más efectivo y cómodo para el usuario.

4.2. GUI_GatewayUSBtin

Esta interfaz gráfica se ha creado para facilitar al usuario la conexión y desconexión del bus CAN, así como la modificación de determinados parámetros de funcionamiento, como por ejemplo el puerto serie, la velocidad del CAN (baudrate), las máscaras y los filtros, entre otros, con una mayor comodidad.

El fichero *GUI_GatewayUSBtin.py* se encuentra en el Anexo 2. Este programa utiliza el módulo Tkinter de python. Esta biblioteca es muy útil y fácil de entender para realizar pequeñas interfaces gráficas como la que se ha desarrollado en este proyecto. Una de las ventajas de Tkinter es que es 100% compatible con cualquier plataforma y sistema operativo.

Como el fichero anterior *(GatewayUSBtin.py),* al principio *(GUI_GatewayUSBtin.py – líneas 1-9)* se encuentran las bibliotecas importadas en dicho módulo. En concreto, se importan cinco bibliotecas nuevas, una de ellas es la biblioteca propia de python Tkinter, de la cual se importan adicionalmente dos módulos específicos *(ttk y messagebox).* Otra es el módulo que contiene la puerta de enlace desarrollada en este proyecto e incluida en el fichero *GatewayUSBtin.py.* Las otras tres son *sys, glob y serial,* que se utilizan para detectar si hay algún dispositivo puerto serie periférico conectado en un puerto COM del ordenador.

Así pues, el programa será principalmente una clase de python, aunque se escriben dos funciones adicionales *(GUI_GatewayUSBtin.py – líneas 13-70)* antes de la clase. Estas funciones son las encargadas de detectar algún dispositivo periférico de comunicación por puerto serie, en este caso, para detectar el dispositivo USBtin. Esta función crea una lista con los puertos seria (COM) disponibles en el ordenador que más adelante, se mostrarán en la interfaz gráfica. En caso de no encontrar ningún dispositivo conectado, el programa mostrará un mensaje de error y no permitirá conectarse al bus CAN.

A continuación se encuentra la clase principal *(GUI_GatewayUSBtin.py – líneas 68-218).* Dentro del *__init__() (GUI_GatewayUSBtin.py – líneas 74-174)* se crea la ventana con el nombre *"GUI_GatewayUSBtin.py"* y cuatro frames [15].

En el primer frame (Fig. 4.2.1) se colocan dos desplegables *"Combobox",* acompañados de dos etiquetas "labels", *"PORT"* y "*BAUD*". Estos desplegables se encuentran en la biblioteca *tkk* mencionada anteriormente y permiten seleccionar el puerto serie disponible para la conexión y la velocidad de transmisión de datos del bus CAN real.

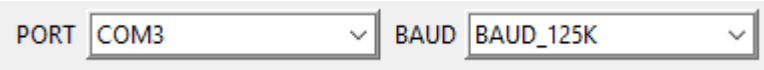


Fig. 4.2.1. Frame 1. Fuente: propia.

En el segundo frame (Fig. 4.2.2.) aparecen dos cajas de texto *"Entry"* con las etiquetas "MASK" y "FILTER". Al ejecutar la interfaz, se colocan unos valores por defecto de máscara y filtro. El valor que se utiliza por defecto de máscara es '0x7FF', de manera que active todos los identificadores posibles y así recibir todo mensaje con identificador comprendido entre '0x000' y '0x7FF' (11 bits).

Fig. 4.2.2. Frame 2. Fuente: propia.

El frame 4 *(Fig. 4.2.3.)* es un botón para conectar o desconectar la puerta de enlace del bus CAN. El botón está asociado a una función dentro de la clase principal *(GUI_GatewayUSBtin.py – líneas 176-197)* llamada *StartStop().* Si el botón está en el estado *'start'*, se configuran la máscara y el filtro y se activa la puerta de enlace *GatewayUSBtin.py.* Por el contrario, si el estado pasa a *'stop'*, se llama a la función *stop() (GatewayUSBtin.py – líneas 100-107)* en la que se cierra el canal del BUS virtual y los dos hilos.

Fig. 4.2.3. Frame 3. Fuente: propia.

Por último, el frame 4 *(Fig. 4.2.4.)* es una etiqueta que informa sobre el estado de la puerta de enlace avisando de si está conectada o desconectada. En la Fig. 4.2.5. se muestra el aspecto que tiene esta pequeña interfaz.

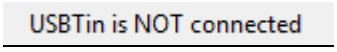


Fig. 4.2.4. Frame 4. Fuente: propia.

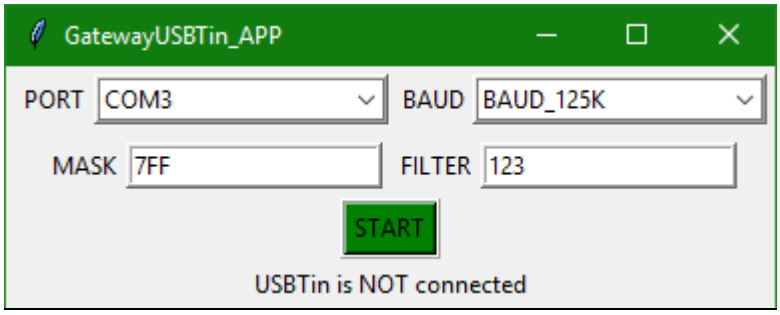


Fig. 4.2.5. GUI_GatewayUSBtin. Fuente: propia.

4.3. Ficheros adicionales (Herramientas)

4.3.1. Biblioteca usbTinLib

Gracias a la biblioteca *usbTinLib.py [10]* (ANEXO 3), se puede acceder al USBtin para leer o transmitir mensajes del bus CAN. Para ser más específicos, dentro de esta biblioteca se utilizarán las siguientes funciones:

canOpenChannel():

Función que abre un canal *(usbTinLib.py – líneas 221-264*). El USBtin tiene tres modos posibles de conexión, *"Normal"," Listen-only"* y *"Loopback"*. En nuestro caso, a lo largo de todo el documento se utiliza el modo *"Normal",* en el cual se puede tanto transmitir mensajes del bus CAN como leerlos.

canReadMessage():

Función capaz de procesar mensajes transmitidos por el bus CAN al que está conectado el USBtin. Este método *(usbTinLib – líneas 525-569*) devuelve una lista con las siguientes cadenas de texto o strings: *"[Modo, Id, longitud, Datos]".*

canWrite(Id, Longitud, Datos, Mascara):

La función canWrite() *(usbTinLib.py – líneas 346-416)* es capaz de transmitir un mensaje desde el USBtin al bus CAN. La función tiene una serie de parámetros de entrada que se deben conocer. Identificador del mensaje (Id), longitud del mensaje (Longitud), datos del mensaje (Datos) y mascara del mensaje (Mascara).

canSetFilter(Filtro):

La función se encuentra en el archivo *(usbTinLib.py – líneas 332-337)*. Es capaz de aplicar un filtro específico para solo recibir mensajes de un identificador o de un grupo de identificadores en concreto, por ejemplo, de un sensor específico conectado a un nodo concreto del bus CAN. Este filtro depende de si la máscara lo activa o no. A continuación, se explica el funcionamiento de la máscara.

canSetMask(Mascara):

Esta función *(usbTinLib.py – líneas 339-344)* activa o no, dependiendo de su valor bit a bit, el filtro del identificador que se desee recibir. Si el valor es "1" se activa el filtro y si el valor es "0" se desactiva el filtro. Ya que USBtin trabaja con máscaras de longitud máxima de 11-bits, si se aplica la máscara '0x7FF' (0b11111111111) se permite la lectura de todos los identificadores posibles. Si, por el contrario, se aplica la máscara '0x000' (0b00000000000), únicamente se aceptarán aquellos mensajes especificados por el filtro escogido.

canClose():

Esta función *(usbTinLib.py – líneas 266-284*), simplemente cierra el canal del bus CAN. Una vez utilizada esta función no se puede acceder al canal sin volverlo a abrir.

closeSerial():

La función *closeSerial*() (*usbTilLib.py – líneas 92-112*) Esta función es similar a la anterior pero más completa, es decir, aparte de cerrar el canal del bus CAN, también cierra el puerto serie y los hilos (véase capítulo 4.3.2). Una buena manera de verificar si se ha ejecutado esta función es visualizar en el Shell de Python la siguiente línea: "*Port is closed*".

A parte de estas funciones que se utilizan directamente desde la puerta de enlace *GatewayUSBtin.py*, dentro del fichero *usbTinLib.py* existen más métodos que son utilizados para un correcto funcionamiento del sistema.

4.3.2. Fichero PeriodicThread

Para conseguir un correcto funcionamiento de la puerta de enlace es necesaria la utilización de programación concurrente. Este tipo de programación, como su propio nombre indica, significa que, en un mismo sitio o momento, se llevan a cabo diferentes procesos. En el caso de este proyecto, se están ejecutando dos funciones a la vez, *LeerMensajesUSBtin()* y *LeerMensajesBusMaster().* Más concretamente, cada función será un "hilo" que se ejecutará al conectarse al bus CAN y se cancelará cuando se desconecte.

Así pues, gracias al fichero *PeriodicThread.py* (Anexo 4) se pueden crear los dos "hilos" comentados anteriormente, que se están ejecutando al mismo tiempo para poder leer tanto

mensajes transmitidos desde el USBtin como desde BusMaster. Los métodos empleados son los siguientes.

PeriodicThread.PeriodicThread()

Crea un 'hilo' que ejecuta continuamente el método que se desee. En el caso de nuestra puerta de enlace, se crean dos 'hilos' (threads). Uno que está constantemente leyendo mensajes transmitidos desde USBtin y otro que está constantemente leyendo los mensajes enviados desde BusMaster. Gracias a este módulo Python, el sistema será bidireccional entre USBtin y BusMaster.

4.3.3. Funciones específicas bus CAN

Una vez vistos las funciones relacionadas directamente con USBtin, se presentan las mismas funciones ligadas a BusMaster y gracias a las cuales, se puede hacer compatibles el software open source BusMaster con nuestro hardware de bajo coste USBtin. La biblioteca que se importa para poder acceder al software, se llama *canlib*. Como se ha explicado anteriormente, ha sido importada en el fichero de trabajo. Así pues, a continuación, se detallan los métodos ligados a la comunicación con BusMaster.

canlib.canlib():

Crea el objeto bus CAN. La forma de iniciarlo debe ser, por ejemplo, '*busCAN1 = canlib.canlib()*'.

openChannel (0, canlib.canOPEN_ACCEPT_VIRTUAL):

Atributo que abre un canal virtual CAN. El primer parámetro '0' indica el número del canal y el segundo es el comando que hay que utilizar para crear un bus virtual. El puntero a dicho canal se guarda en otra variable como '*ch = c1.openChannel(0, canlib.canOPEN_ACCEPT_VIRTUAL)'.*

busOn():

Atributo que activa el bus. Es completamente necesario realizar esta función antes de leer o escribir mensajes en el bus. A diferencia del atributo anterior, ahora no será necesario crear una nueva variable. Se utiliza escribiendo en una línea el siguiente comando '*ch.busOn()'.*

read():

Atributo que como su propio nombre indica, permite leer mensajes del bus CAN virtual. Esta herramienta te devuelve una lista con diferentes datos del mensaje. Así pues, del mensaje se puede distinguir entre su id, el tipo de mensaje, longitud, tiempo y el significado en sí. En este caso, se utilizará una lista para guardar el mensaje y así poderlo procesar posteriormente, como por ejemplo, representándolo en el Shell de python. Un mensaje se lee de la siguiente forma: *'msgId, data, dlc, flags, time = ch.read()'.*

write():

Éste se utiliza de forma bastante similar al *read(),* la única diferencia es que el mensaje (lista con los datos propios del mensaje) no se tiene que guardar en una lista, debido a que para escribir un mensaje se entiende que ya lo tienes guardado con algún comando usado anteriormente. La forma de escribir un mensaje es: *'ch.write(msgId, data, dlc, flags, time)'.*

busOff():

Función análoga a *busOn(),* que se encarga de desactivar el canal del bus virtual, de manera que no se pueda ni leer ni escribir mensajes. Se utiliza de igual manera que *busOn(),* es decir, una línea con el siguiente comando *'ch.busOff()'.*

close():

Por último, esta función cierra el canal del bus virtual por completo y se utiliza de la siguiente manera '*ch.close()*'.

4.4. Resultados

En este apartado se presentan los resultados obtenidos utilizando diferentes tipos de mensaje, máscaras y filtros. Para verificar el correcto funcionamiento del programa se utiliza el dispositivo Kvaser HS Leaf Light y el software CanKing, desde el cual se enviarán los mensajes al bus CAN desde un ordenador secundario. En el PC principal se conecta un USBtin y se ejecutan, tanto el BusMaster, como la interfaz de la puerta de enlace desarrollada en este proyecto.

Los test realizados para verificar el funcionamiento de la puerta de enlace son los siguientes:

- Recepción de mensajes de un bus CAN real sin aplicar ningún filtro.
- Recepción de mensajes de un bus CAN real aplicando un filtro.
- Recepción de mensajes de un bus CAN real utilizando identificadores standard y extendido.
- Recepción de mensajes de un bus CAN real a diferentes velocidades de transferencia (Baudrate).
- Modificación del puerto serie al que está conectado el USBtin.
- Envío de mensajes desde BusMaster.
- Monitorización en forma de gráfico de dos variables contenidas en un mensaje de CAN.

4.4.1. Recepción de mensajes CAN sin filtro

En este test se aplica la máscara en formato hexadecimal '7FF' al USBtin. Este valor de máscara no aplica ningún filtro al identificador del mensaje, por lo tanto, se deben poder leer todos los mensajes del bus CAN y mostrarlos en BusMaster.

A continuación, *(Fig. 4.3.1.1)* se observa cómo se configura la transmisión de un mensaje desde CanKing para poderlo enviar al bus CAN real desde el dispositivo Kvaser HS Leaf Light de Kvaser.

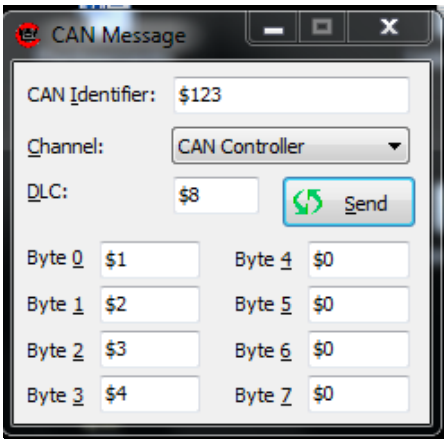


Fig. 4.3.1.1 Ejemplo de mensaje en CanKing. Fuente: propia.

Para la simulación se envían mensajes con diferentes identificadores y datos. Estos mensajes enviados desde CanKing se pueden ver en la *Fig. 4.3.1.2.*

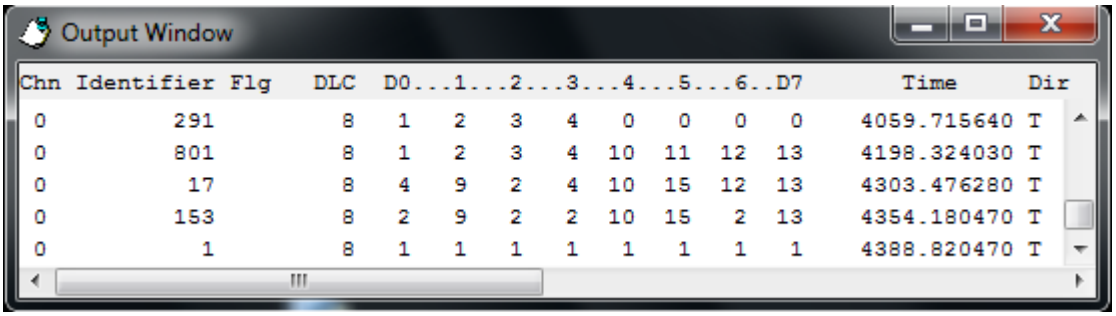


Fig. 4.3.1.2 Output window de CanKing. Fuente: propia.

En la siguiente figura *(Fig. 4.3.1.3.)* se observan los valores de la máscara y el filtro. Para realizar una simulación sin filtro se debe colocar el número '*0x7FF'*. De esta manera, el filtro queda desactivado y deja pasar todos los identificadores posibles comprendidos entre *'0x000' y 0x7FF'*.

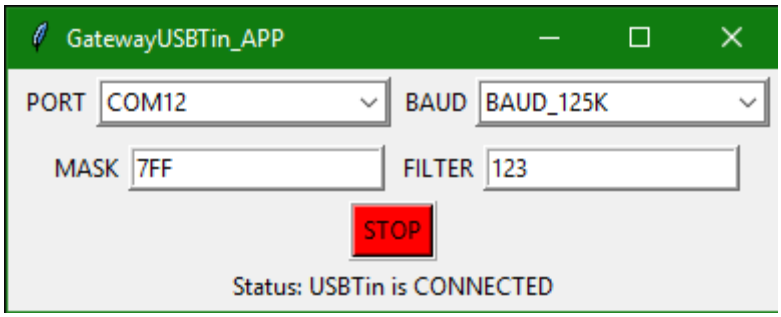
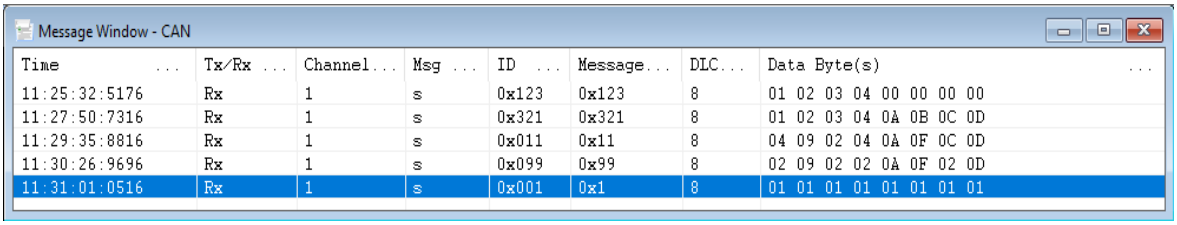


Fig. 4.3.1.3. Valores GatewayUSBtin. Fuente: propia.

Por último, en la siguiente *Fig. 4.3.1.4.* se observan los mensajes recibidos por el USBtin en el software BusMaster. Como se ha dicho anteriormente, al no activar ningún filtro se recibe la totalidad de los mensajes.



Time	Tx/Rx	Channel	Msg	ID	Message	DLC	Data Byte(s)
11:25:32:5176	Rx	1	s	0x123	0x123	8	01 02 03 04 00 00 00 00
11:27:50:7316	Rx	1	s	0x321	0x321	8	01 02 03 04 0A 0B 0C 0D
11:29:35:8816	Rx	1	s	0x011	0x11	8	04 09 02 04 0A 0F 0C 0D
11:30:26:9696	Rx	1	s	0x099	0x99	8	02 09 02 02 0A 0F 02 0D
11:31:01:0516	Rx	1	s	0x001	0x1	8	01 01 01 01 01 01 01 01

Fig. 4.1.3.4. Output window de BusMaster. Fuente: propia

4.4.2. Recepción de mensajes CAN con filtro

En este test se activa el filtro del USBtin para recibir únicamente mensajes CAN de un identificador en concreto. Un ejemplo de aplicación de un filtro en un bus CAN de un coche, por ejemplo, es recibir únicamente los mensajes de un sensor concreto en el coche del que se están recibiendo continuos mensajes de error y por lo tanto, se debe estudiar, en concreto, solo este sensor.

Como en el test anterior, se presentan los siguientes resultados. En la *Fig. 4.3.2.1.* se pueden ver los mensajes enviados al bus CAN real desde el dispositivo HS Leaf Light de Kvaser.

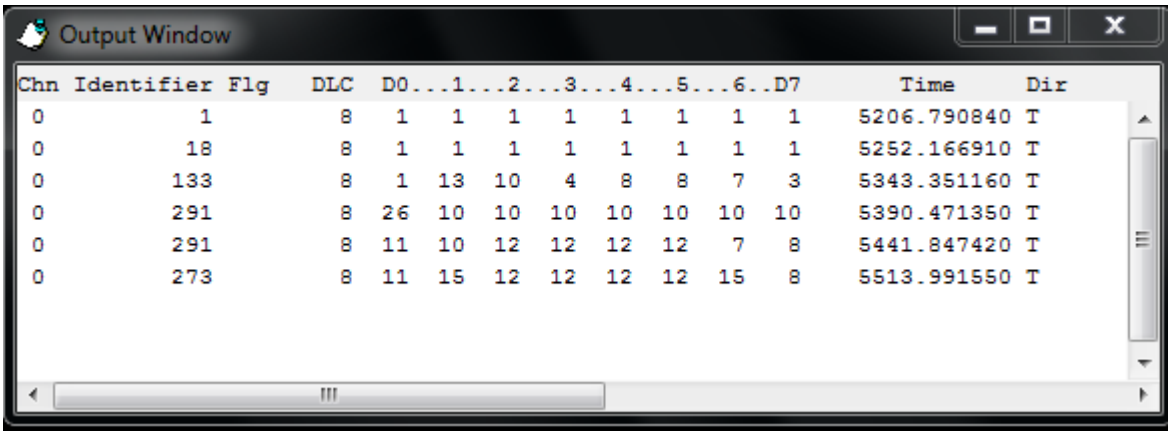


Fig. 4.3.2.1. Output window de CanKing. Fuente: propia.

Seguidamente, se muestran en la *Fig. 4.3.2.2.* los datos correspondientes de máscara y filtro en la puerta de enlace *GatewayUSBtin.* Cambiando el valor de la máscara por *'0x000'*, se activa por completo el filtro. Por lo tanto, si escogemos para el filtro el identificador *'0x123'*, BusMaster solo recibirá mensajes con dicho identificador.

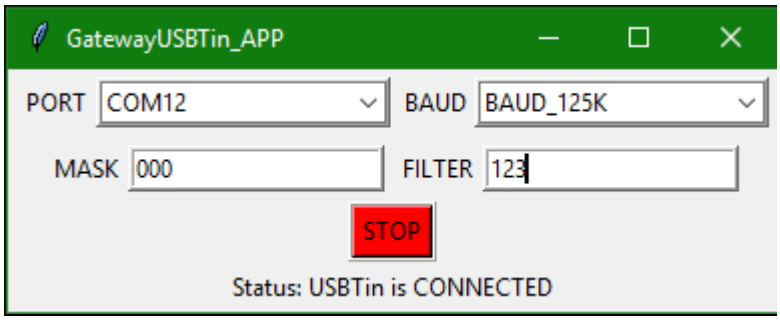


Fig. 4.3.2.2. GUI_GatewayUSBtin. Fuente: propia.

Por último, en la *Fig. 4.3.2.3.* se observan los mensajes recibidos por BusMaster.

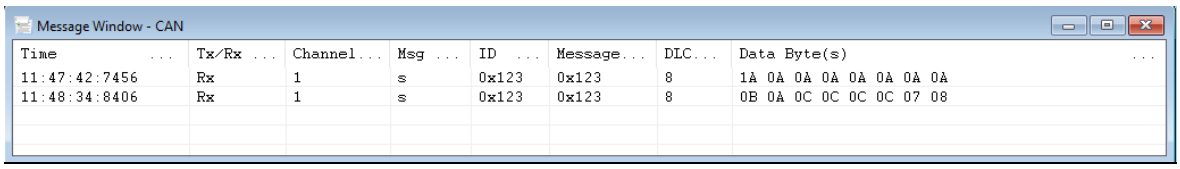


Fig. 4.3.2.3. Output window de BusMaster. Fuente: propia.

Como se puede observar, solo se han recibido dos mensajes con el identificador '0x123' de los seis enviados. El cuarto y quinto mensajes de los enviados (Fig. 4.3.2.1.) son los que ha recibido BusMaster.

*NOTA: El *'output window de CanKing' (Fig. 4.3.2.1.)* expresa los datos en sistema decimal

mientras que en el *'output window de BusMaster' (Fig. 4.3.2.3.)* expresa los datos en sistema hexadecimal.

4.4.3. Tipo de mensaje

Para realizar este test, se utilizan dos tipos de mensajes, estándar (Std) y extendido (Ext). Normalmente en los turismos se utilizan mensajes estándar, aunque, si es necesario, se pueden usar mensajes extendidos, como por ejemplo, en los camiones y autobuses. Un mensaje standard es un mensaje de 11 bits de longitud, cuyo identificador puede variar entre los valores hexadecimales *'0x000' y '0x7FF'*. De forma análoga, un mensaje extendido (21 bits) puede variar su identificador en el rango hexadecimal *'0x0000000' – '0x1FFFFFFF'.*

A continuación, en las *figuras (4.3.3.1 – 4.3.3.2 – 4.3.3.3 – 4.3.3.4 – 4.3.3.5),* se presentan los resultados de los test anteriores.

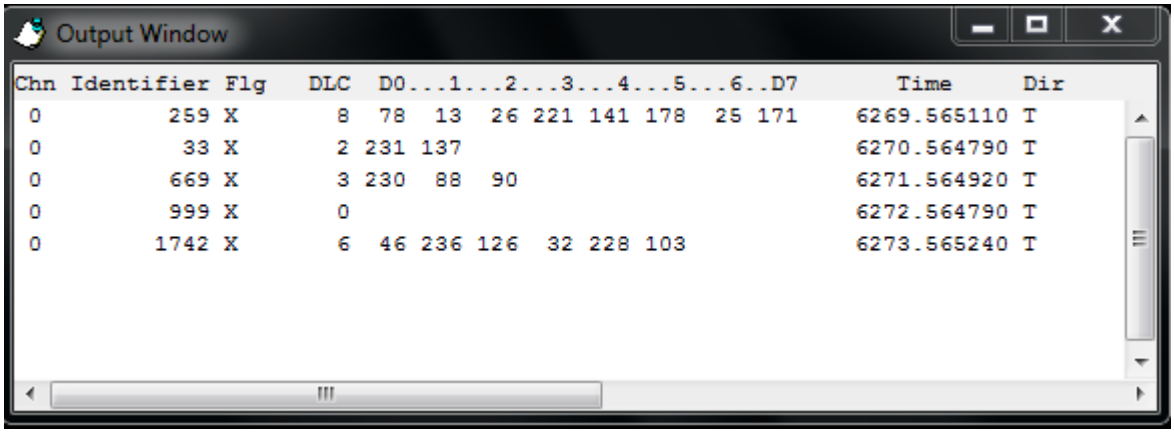


Fig. 4.3.3.1. Output window de CanKing con mensajes extendidos. Fuente: propia.

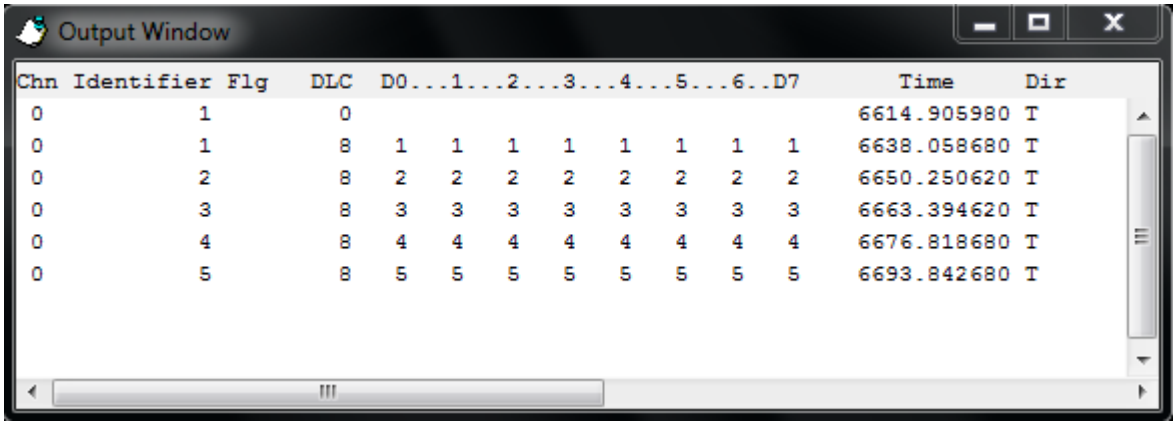


Fig. 4.3.3.2. Output window de CanKing con mensajes estándar. Fuente: propia.

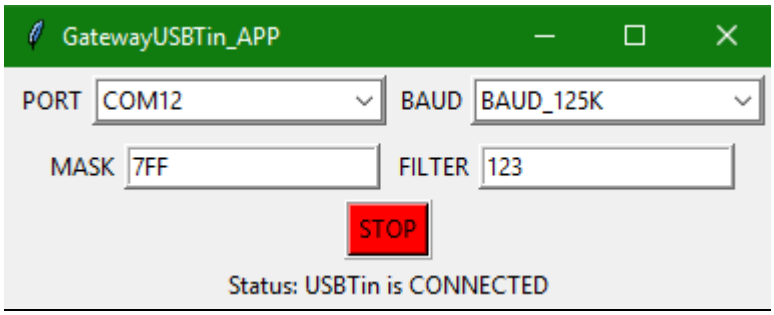


Fig. 4.3.3.3. GUI_GatewayUSBtin. Fuente: propia.

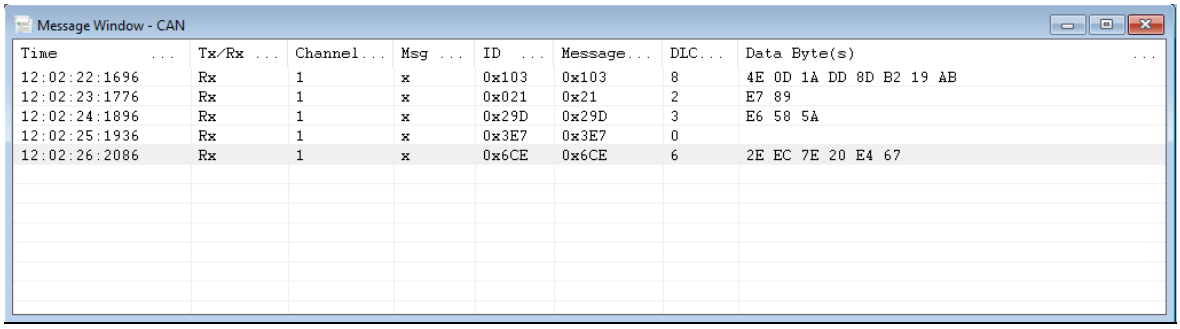


Fig. 4.3.3.4. Output window de BusMaster con mensajes extendidos. Fuente: propia.

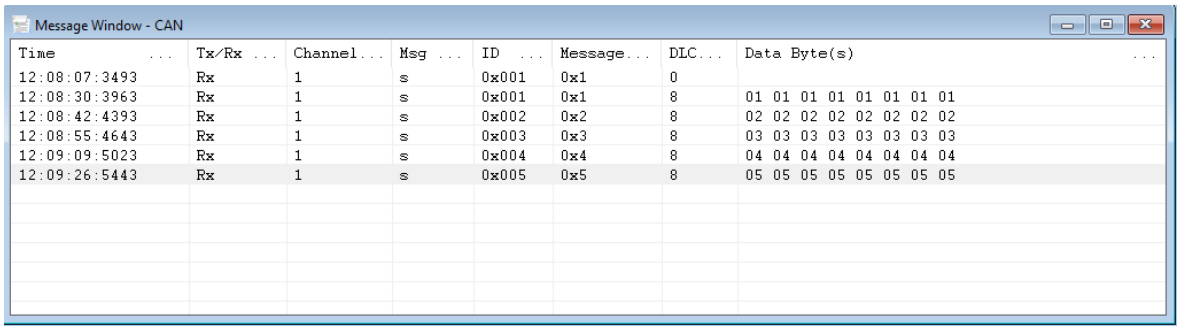


Fig. 4.3.3.5. Output window de BusMaster con mensajes estándar. Fuente: propias.

4.4.4. Variación de la velocidad de trasmisión de datos (Baudrate)

En este caso, se desconecta la puerta de enlace, el BusMaster y el USBtin para así poder cambiar la velocidad. Se utiliza una nueva velocidad (baudrate de CAN) de 250 Kb/s y se vuelve a conectar. El filtro se deja desactivado para recibir todos los mensajes.

Los resultados obtenidos son los siguientes.

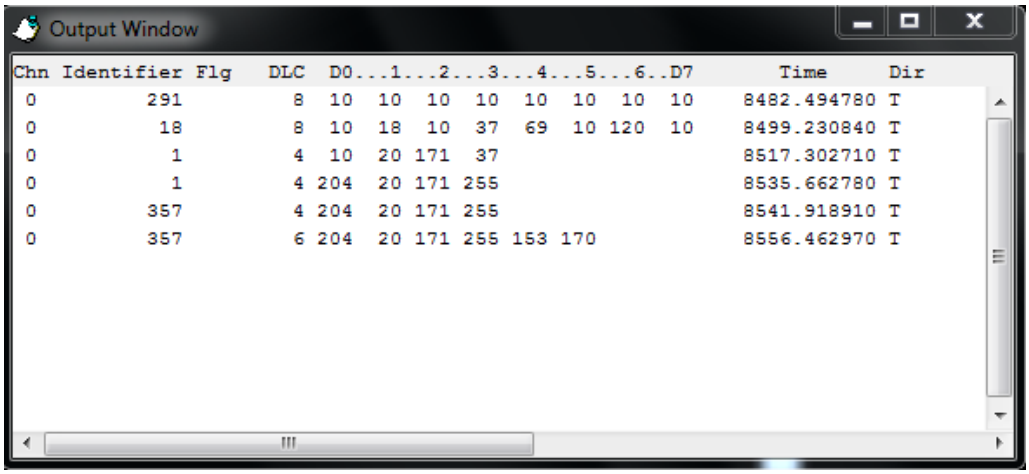


Fig. 4.3.4.1. Output window de CanKing. Fuente: propia.

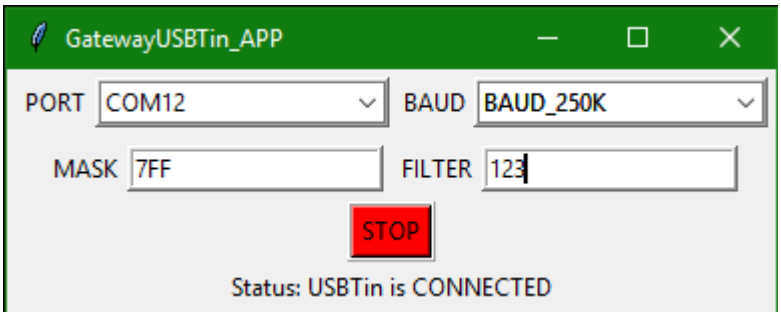


Fig. 4.3.4.2. GUI_GatewayUSBtin. Fuente: propia.

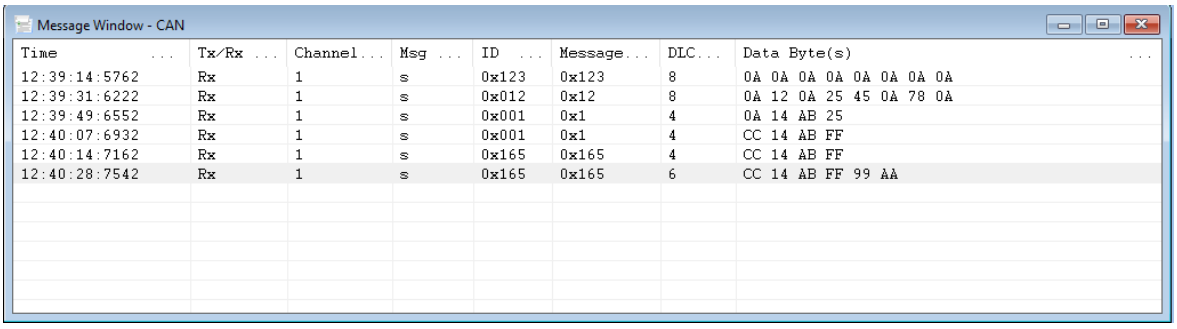


Fig. 4.3.4.3. Output window de BusMaster. Fuente: propia.

Con esta nueva velocidad o baudrate, se prueba a activar el filtro colocando en la máscara el valor *'0x000'* y en el filtro el identificador *'0x123'*. Los resultados son los siguientes.

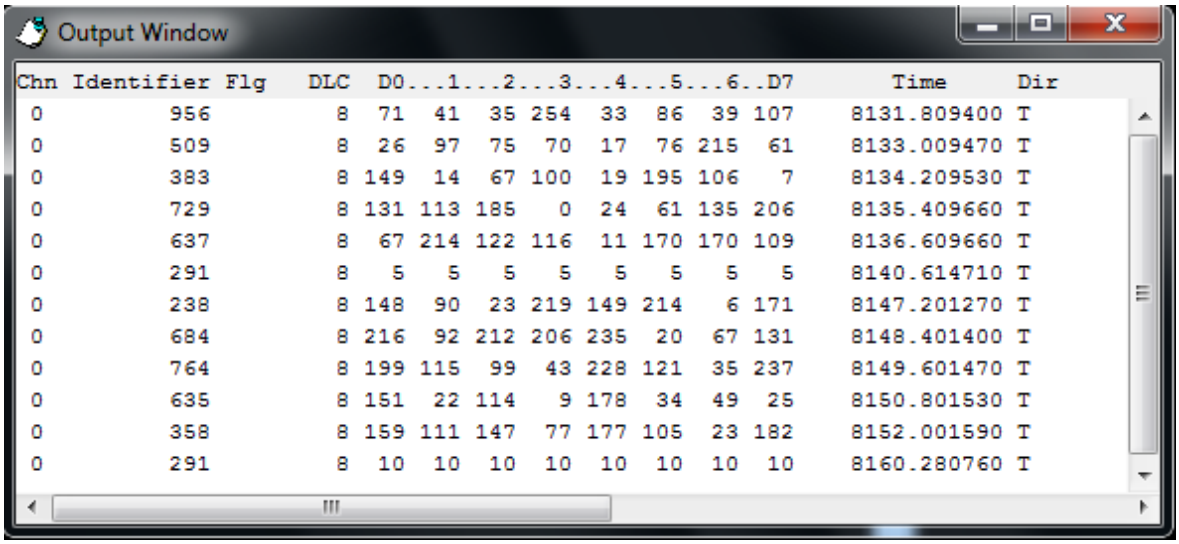


Fig. 4.3.4.4. Output window de CanKing. Fuente: propia.

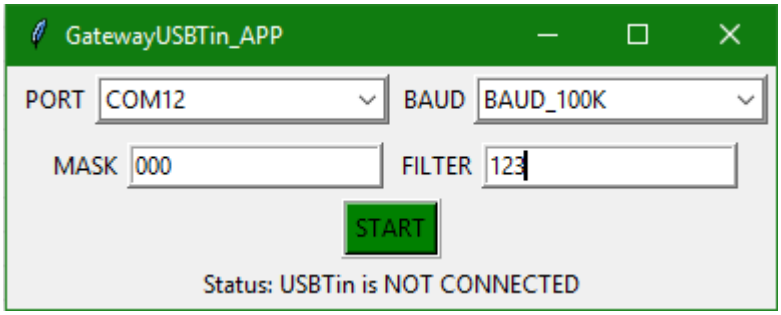


Fig. 4.3.4.5. GUI_GatewayUSBtin. Fuente: propia.

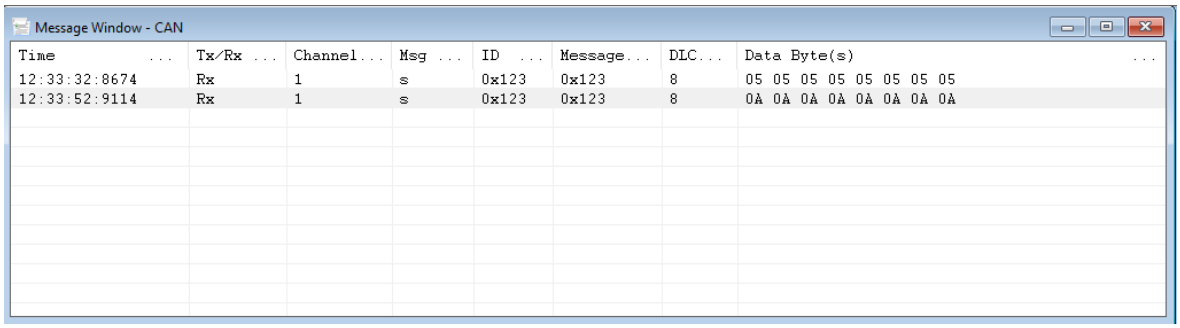


Fig. 4.3.4.6. Output window de BusMaster. Fuente: propia.

*NOTA: El *'output window de CanKing' (Fig. 4.3.4.4.)* expresa los datos en sistema decimal mientras que en el *'output window de BusMaster' (Fig. 4.3.4.6.)* expresa los datos en sistema hexadecimal.

4.4.5. Puerto Serie

Seguidamente, se realiza un último test experimental para verificar que verdaderamente se detectan todos los puertos serie COM del ordenador. Para ello, se desconecta la puerta de enlace y seguidamente cambiamos de puerto físico USB el dispositivo USBtin para volverlo a conectar en otro puerto físico. En este ensayo cambiaremos el dispositivo Kvaser por otro dispositivo USBtin y se utiliza la interfaz gráfica *USBtinViwer_v1.3*. Los resultados son los siguientes.

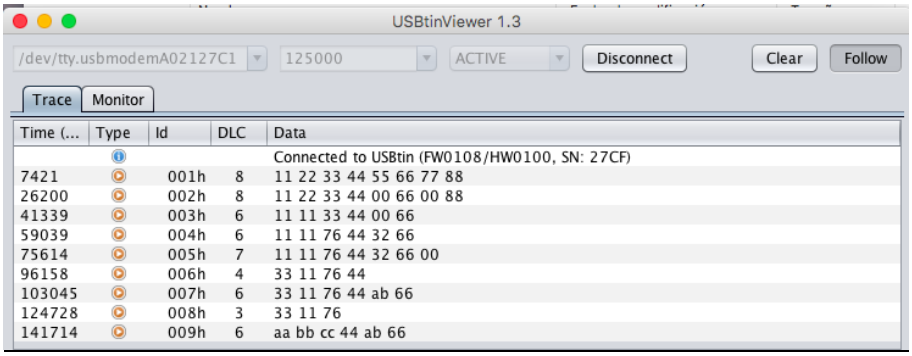


Fig. 4.3.5.1. Output window de USBtinViwer_v1.3. Fuente: propia.

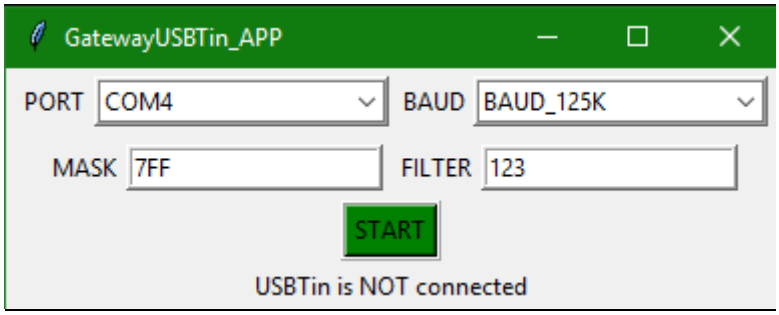


Fig. 4.3.5.2. GUI_GatewayUSBtin. Fuente: propia.

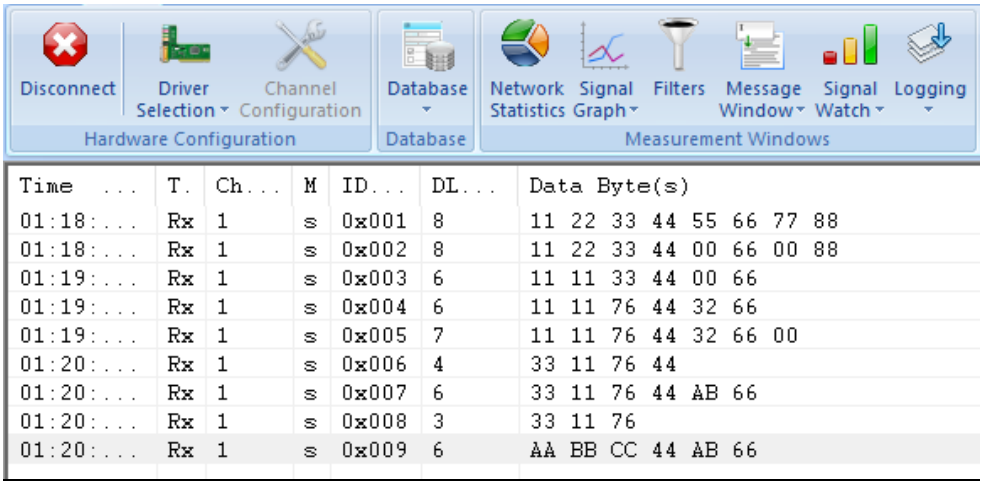
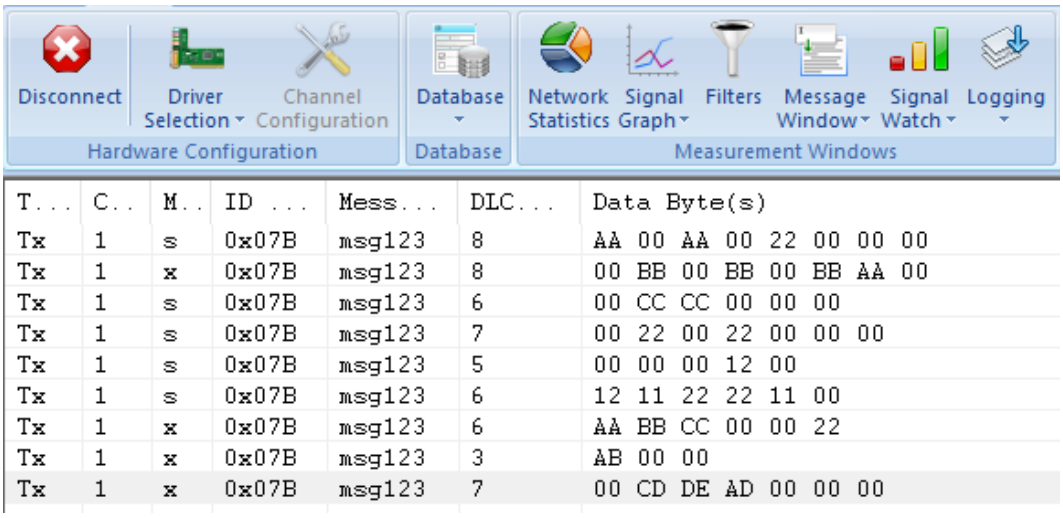


Fig. 4.3.5.3. Output window de BusMaster. Fuente: propia.

4.4.6. Transmisión de mensajes desde BusMaster

Para finalizar, se prueba el camino inverso, es decir, enviar mensajes desde BusMaster para que lleguen al bus CAN real (físico). Este camino es menos importante que el otro, puesto que el principal objetivo de esta puerta de enlace es poder mostrar desde BusMaster los mensajes recibidos por un USBtin. No obstante, la puerta de enlace esta capacitada para poder transmitir mensajes desde BusMaster.

En las siguientes capturas se pueden ver los mensajes enviados desde BusMaster al bus CAN virtual y gracias a la puerta de enlace, recibidos en el USBtin para que lleguen al bus CAN real. Los mensajes son de tipo standard y extendido.



T...	C..	M..	ID ...	Mess...	DLC...	Data Byte(s)
Tx	1	s	0x07B	msg123	8	AA 00 AA 00 22 00 00 00
Tx	1	x	0x07B	msg123	8	00 BB 00 BB 00 BB AA 00
Tx	1	s	0x07B	msg123	6	00 CC CC 00 00 00
Tx	1	s	0x07B	msg123	7	00 22 00 22 00 00 00
Tx	1	s	0x07B	msg123	5	00 00 00 12 00
Tx	1	s	0x07B	msg123	6	12 11 22 22 11 00
Tx	1	x	0x07B	msg123	6	AA BB CC 00 00 22
Tx	1	x	0x07B	msg123	3	AB 00 00
Tx	1	x	0x07B	msg123	7	00 CD DE AD 00 00 00

Fig. 4.4.6.1. Output window de BusMaster. Fuente: propia.

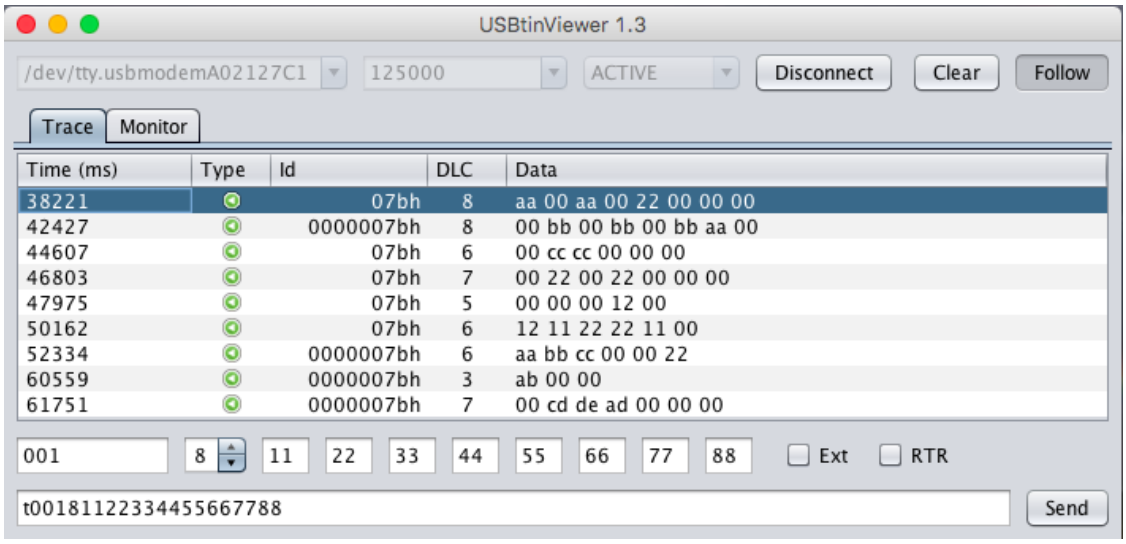


Fig. 4.4.6.2. Output window de USBtinViwer. Fuente: propia.

4.4.7. Gráficos BusMaster a tiempo real

En este último test, diferente a los demás, se programa la transmisión de un conjunto de mensajes CAN que se envían al bus de manera que el USBtin conectado al ordenador pueda recibirlos. Es decir, el montaje es exactamente el mismo que en los anteriores test, aunque en este no va a ser necesario el uso del dispositivo Kvaser HS Leaf Light ni de la interfaz gráfica propia de USBtin *(USBtinViwer).*

En este caso, se utilizan dos USBtin conectados a ordenadores diferentes y, entre ellos, por el bus CAN. Desde un ordenador, gracias a los métodos que ofrece la biblioteca *usbTinLib.py,* se programan los mensajes que son enviados desde el dispositivo USBtin conectado a dicho ordenador. Estos mensajes se reciben desde el otro dispositivo y gracias a la puerta de enlace que está activada en el otro PC, se puede observar de forma gráfica los mensajes anteriormente programados.

Para enviar el paquete de mensajes CAN, se utiliza un programa simple escrito en lenguaje Python. Se programa el envío de dos señales triangulares, una será las revoluciones por minuto (RPM) del motor de un coche y la otra el voltaje. Ambas señales se enviarán en un mismo mensaje y se podrán graficar simultáneamente en BusMaster.

El valor de las RPM simuladas está basado en una señal triangular periódica que oscila entre 500 y 1000 rpm. Por otro lado, el voltaje también es una señal triangular cíclica de menor amplitud que las RPM. Este voltaje varía entre 10 y 100 Voltios. Ambas señales están sincronizadas, de manera que cuando la pendiente es positiva para una, para la otra también y de forma análoga, si la pendiente es negativa. Esta señal se programa gracias a herramientas básicas como un bucle (while) que se ejecuta indefinidamente hasta que el usuario lo cancele. Este programa se coloca al final del fichero (*__main__)* para así utilizar las funciones que interese como *canOpenChannel(), canWrite() o closeSerial().*

A continuación *(Fig. 4.4.7.1),* se presenta el código añadido al final del fichero *usbTinLib.py.* El código se ubica en la ultima parte del fichero, dentro del __main__ *(if __name__ == "__main__":).* Así pues, el nuevo *__main__* de la clase *USBtin()* queda de la siguiente manera:

*NOTA: Destacar que este fichero se debe duplicar, modificar y guardar en un fichero aparte para así no confundirlo con el anterior y evitar cualquier tipo de problema posterior.

```
if __name__ == "__main__":

    import time
    app=USBTin('COM37','BAUD_125K','Normal')
    app.canOpenChannel()
    up = True
    voltage_step = 10
    voltage = 0
    rpm = 500
    rpm_step = 50
    try:
        while True:
            lista =[]
            if up==True:
                voltage = voltage + voltage_step
                rpm = rpm + rpm_step
                if voltage>99:
                    up = False
            else:
                voltage = voltage - voltage_step
                rpm = rpm - rpm_step
                if voltage <11:
                    up = True

            lista.append(format(voltage,'02x'))
            lista.append(format(rpm&0xFF,'02x'))
            lista.append(format(rpm>>8,'02x'))
            app.canWrite('7b',3,lista,0x0002)
            time.sleep(0.25)
    except KeyboardInterrupt:
        app.canClose()
    print("End")
```

Fig. 4.4.7.1. __main__ usbTinLib.py modificado para generar las funciones. Fuente: propia.

Como se observa al principio del *__main__()*, se debe modificar manualmente el puerto COM y la velocidad (Baudrate) de conexión del USBtin. Una vez escogidas estas dos opciones ya está preparado el fichero *usbTinLib.py* para ser ejecutado.

Antes de enviar los mensajes, se prepara el otro PC conectando tanto la puerta de enlace *GUI_GatewayUSBtin.py* como el software BusMaster, ambos a la misma velocidad de transferencia de datos de bus CAN que el otro dispositivo USBtin. Dentro de BusMaster, se utiliza la base de datos que viene por defecto. Una vez asociada, dentro del menú de herramientas *"Measurement windows"* > *"Signal graph"* se configuran ambas señales (RPM y voltaje) y por último se activa el gráfico.

Los resultados obtenidos se presentan en las siguientes figuras. La *Fig. 4.4.7.2.* muestra únicamente el voltaje, en la *Fig. 4.4.7.3.* se puede observar únicamente las RPM y, por último, en la *Fig. 4.4.7.5.* se pueden ver ambos gráficos simultáneamente.

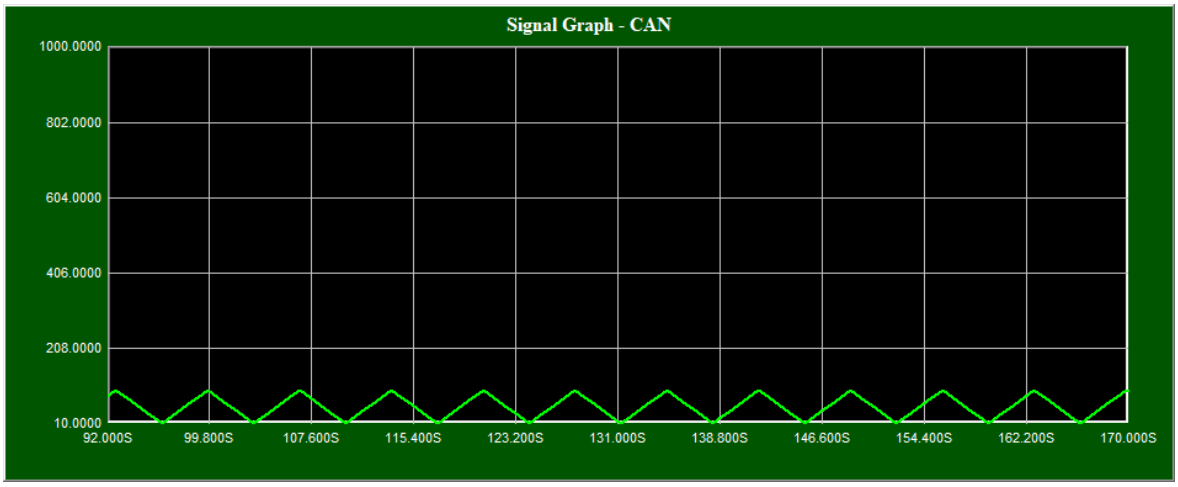


Fig. 4.4.7.2. *Voltaje.* Fuente: propia.

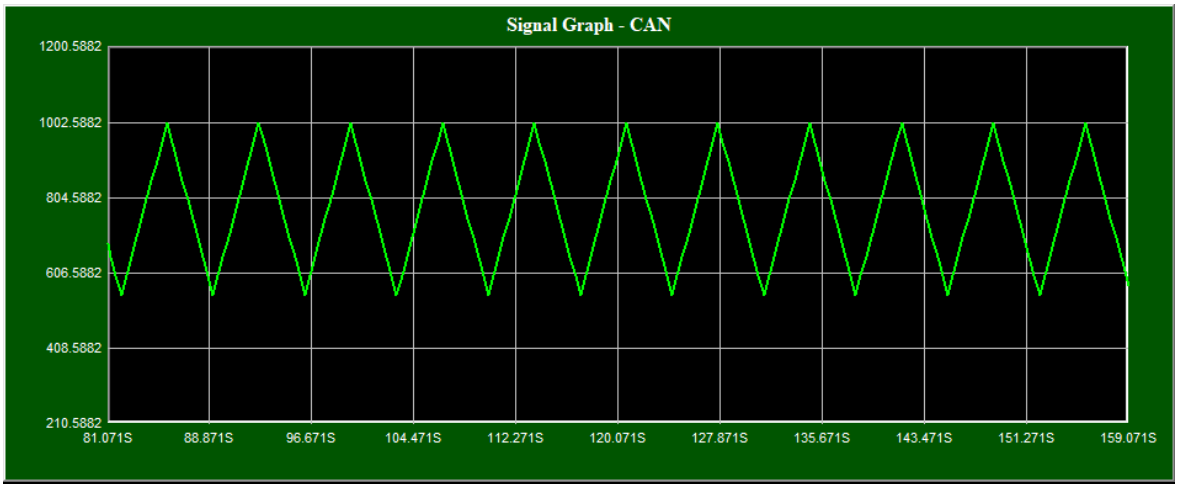


Fig. 4.4.7.3. RPM. Fuente: propia.

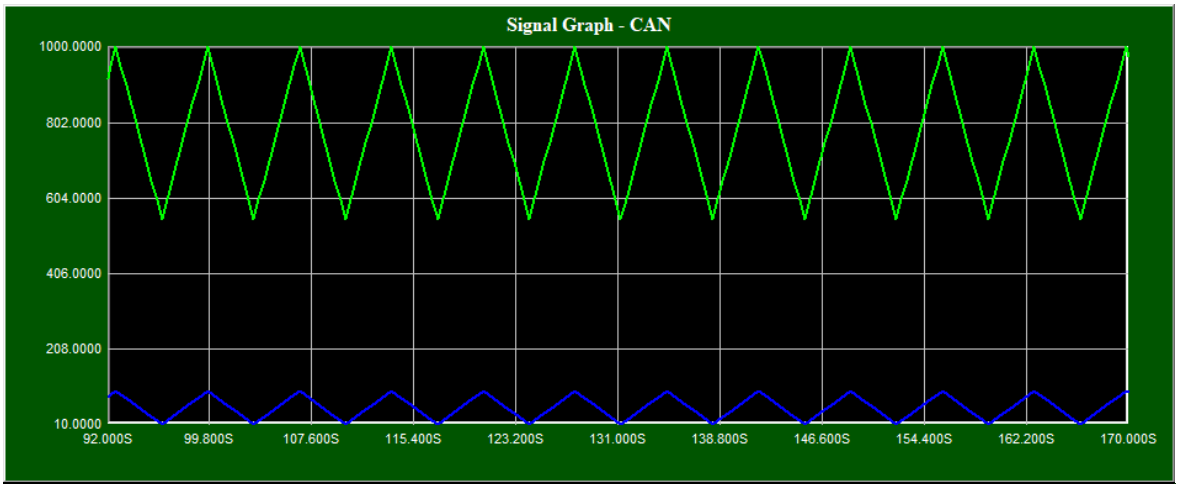


Fig. 4.4.7.4. Voltaje y RPM. Fuente: propia.

5. Programa

Para concluir, se explican los dos programas que se ejecutan en cada "hilo" de la puerta de enlace *GatewayUSBtin.py.* Antes de crear el programa conjunto se verificó el funcionamiento de las dos funciones básicas de la puerta de enlace. Estas funciones son *LeerMensajesBusMaster()* y *LeerMensajesUSBtin.py.* Así pues, a continuación, se explican los comandos básicos para poderse conectar tanto a BusMaster como al USBtin desde Python.

5.1. Mensajes BusMaster

Se estudian las diferentes herramientas que hay en python para conectarse a un bus CAN. BusMaster, como ya se ha comentado anteriormente, trabaja con Kvaser. Desde la página oficial de Kvaser [4] se puede descargar el SDK canlib, una extensión de python necesaria para poder leer mensajes enviados al bus desde BusMaster.

A continuación, se presenta el fichero (Programa_001_BucleRetornarMensajesBusMaster.py), en el cual se crea un bucle que está continuamente leyendo del bus virtual y devolviendo todos los mensajes que pasen por el bus. De esta manera, se crea la primera parte del programa GatewayUSBtin, es la que enlaza BusMaster con la puerta de enlace en python. Para comprobar su correcto funcionamiento, ejecutaremos el programa y conectaremos el BusMaster al mismo canal virtual con la misma velocidad de transferencia de bits [Bits/s]. Si el programa funciona, se deberá repetir el mensaje en BusMaster (uno enviado y otro recibido) y también de verá en el Shell de Python.

En este fichero únicamente se importa el módulo canlib que, si está instalado correctamente en el ordenador, no se hace referencia a ninguna otra biblioteca. Por lo tanto, se puede ejecutar desde cualquier directorio del ordenador y no dará ningún problema.

A continuación, se presenta el código utilizado para este primer programa y unas capturas de pantalla con un ejemplo de mensaje enviado desde BusMaster, procesado por el programa y enviado de nuevo al bus CAN virtual.

Programa_001_bucleRetornarMensajesBusMaster.py

```
import canlib.canlib as canlib

cl = canlib.canlib()
ch = cl.openChannel(0, canlib.canOPEN_ACCEPT_VIRTUAL)
ch.busOn()

while True:
    try:
        msgId, data, dlc, flags, time = ch.read()
        ch.write(msgId, data, flags, dlc)
        print(msgId, data, flags, dlc,time)

    except(canlib.canNoMsg) as ex:
        pass

    except (canlib.canError) as ex:
        print(ex)

ch.busOff()
ch.close()
```

Una vez ejecutado BusMaster, como se observa en la Fig. 5.1.1, se selecciona el canal virtual, en nuestro caso el canal 0 (Channel 0) y su velocidad de transferencia, (125 kb/s).

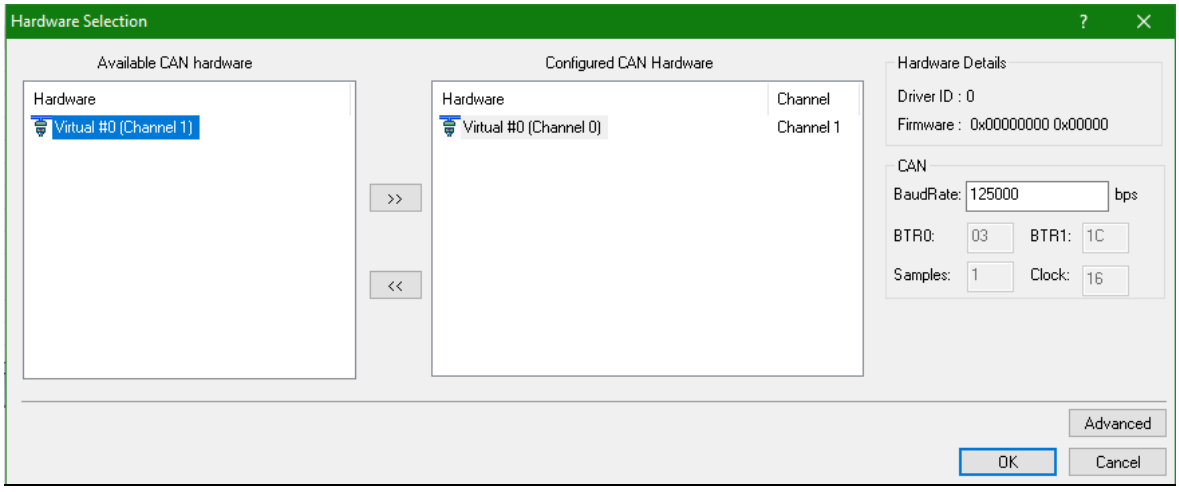


Fig. 5.1.1. Configuración del canal y velocidad del bus. Fuente: propia.

A continuación, después de configurar el canal y la velocidad, se debe asociar una base de datos (*.dbf) para poder configurar un mensaje. Esta base de datos viene por defecto en el paquete de instalación de BusMaster, que se puede encontrar en el siguiente directorio *C:\Program Files (x86)\BUSMASTER_v3.2.2\Examples\TestAutomation.* Una vez asociada la base de datos, se configura un mensaje de ejemplo como el que se presenta a

continuación.

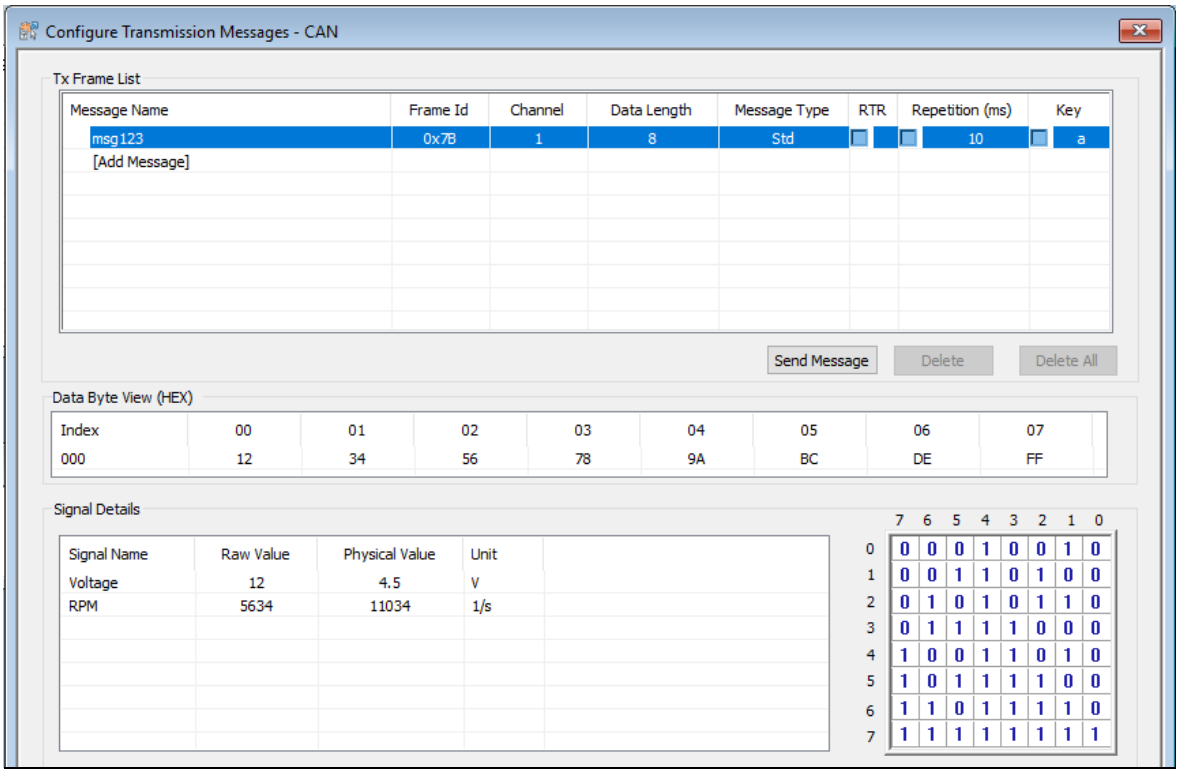


Fig. 5.1.2. Configuración del mensaje en BusMaster. Fuente: propia.

El mensaje enviado presenta las siguientes características:

- **Nombre del mensaje:** msg123
- **Frame Id:** El identificador del mensaje proporciona información sobre el origen de este mensaje. En un bus CAN, se conectan diferentes tipos de sensores y, gracias al identificador de casa mensaje, se sabe de qué sensor proviene el mensaje. En la Fig. 5.1.2 se puede ver que el identificador es *0x7B*.
- **Longitud del mensaje**: Normalmente se utiliza una longitud de 8...
- **Tipo de mensaje (Flags):** Principalmente hay dos tipos de mensajes, estándar o extendido. Se suele utilizar mensajes estándar que son de longitud 11 bits. Si es necesario se puede alargar el identificador y pasa a ser un mensaje extendido (29 bits). Como se puede observar en la Fig. 5.1.2 el tipo de mensaje es estándar (Std).
- **Datos:** Los datos son la información codificada que transporta el mensaje. Como la longitud es 8, el mensaje se divide en 8 bytes. Se expresan en formato hexadecimal, agrupados en 8 números de 2 caracteres cada uno (hexadecimal). En la Fig. 5.1.2 se puede comprobar que el mensaje completo es *'12 34 56 78 9A BC*

DE FF'

Una vez configurado el mensaje, ya se puede conectar al bus CAN virtual desde BusMaster. Por otro lado, ejecutamos el programa desde Python de manera que estén ambos conectados al mismo tiempo. A continuación, enviamos el mensaje desde BusMaster y como observamos en la Fig. 5.1.3, el mensaje pasa por el programa (Programa_001_BucleRetornarMensajes.py) y devuelve el mensaje a BusMaster de nuevo, tal y como se observa en la Fig. 5.1.4. En esta última figura, se pueden apreciar dos mensajes, cuya única diferencia es que el primero está enviado desde BusMaster (Tx) y el último se ha recibido en BusMaster (Rx).

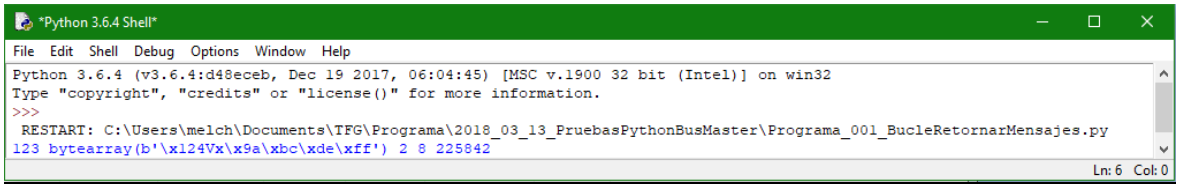


Fig. 5.1.3. Mensaje de prueba en python. Fuente: propia.

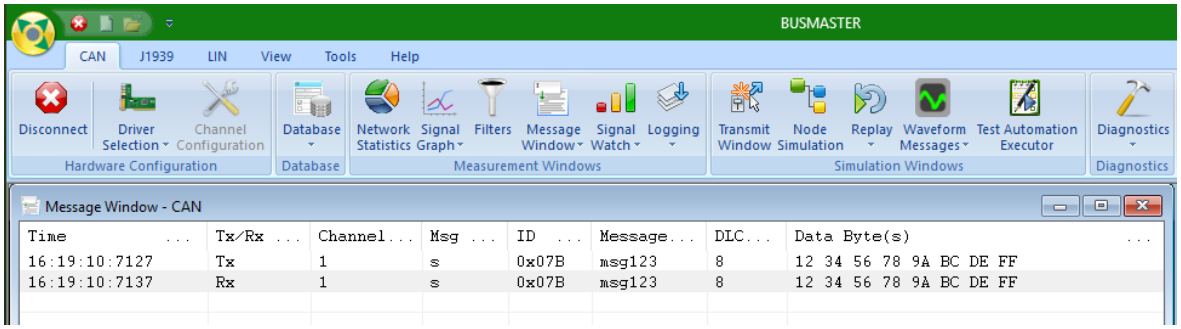


Fig. 5.1.4. Mensaje enviado y recibido en BusMaster. Fuente: propia

Así pues, se corrobora el correcto funcionamiento de este primer programa. A continuación, se crea otra función capaz de leer mensajes del USBtin y, con la ayuda de una herramienta muy útil de Python, los "Threads" (hilos), se juntarán ambos programas en una misma clase en la que trabajarán los dos a la vez. Sin más detalles, se pasa al siguiente apartado para profundizar en el siguiente programa.

5.2. Mensajes USBtin

En este apartado se crea un programa muy simple, en el que se detectan mensajes recibidos desde un dispositivo USBtin. Para este test se utiliza un montaje similar al que se utiliza para los test finales. Con la ayuda de dos dispositivos USBtin, uno envía y el otro recibe. El objetivo de este programa *(Programa_002_TestUSBtin.py)* es recibir desde Python los mensajes recibidos por un USBtin en concreto, en este caso, como se observa en el código del programa, se conecta al puerto de comunicación serie "COM4".

Programa_002_TestUSBtin.py

```
import usbTinLib as usb
import time
import constantsUSB as const

usb1=usb.USBTin('COM4','BAUD_125K','Normal')
print(usb1.askHWversion())
print(usb1.askFWversion())

usb1.canOpenChannel()

while True:
    result = usb1.canReadMessage()
    if result[0]!= const.canERR_NOMSG:
        print(result)

usb1.canClose()
```

El programa simplemente se conecta al USBtin y ejecuta un bucle en el que constantemente está preguntando al USBtin si recibe o no los mensajes. En caso de haber un mensaje, el programa muestra el mensaje recibido por el USBtin en la ventana de Python. Este programa es la base del segundo "hilo" (Thread) que se ejecuta en el fichero principal GatewayUSBtin.py.

A continuación se presentan los resultados obtenidos. En la primera figura (Fig. 5.2.1.) se observan los mensajes enviados al bus CAN desde un dispositivo USBtin conectado al puerto serie "COM3". Por otro lado, en la siguiente figura (Fig. 5.2.2.) se presentan los mensajes recibidos por el programa y representados en la Shell de Python. Los resultados obtenidos son los correctos, ya que se reciben el 100% de mensajes enviados.

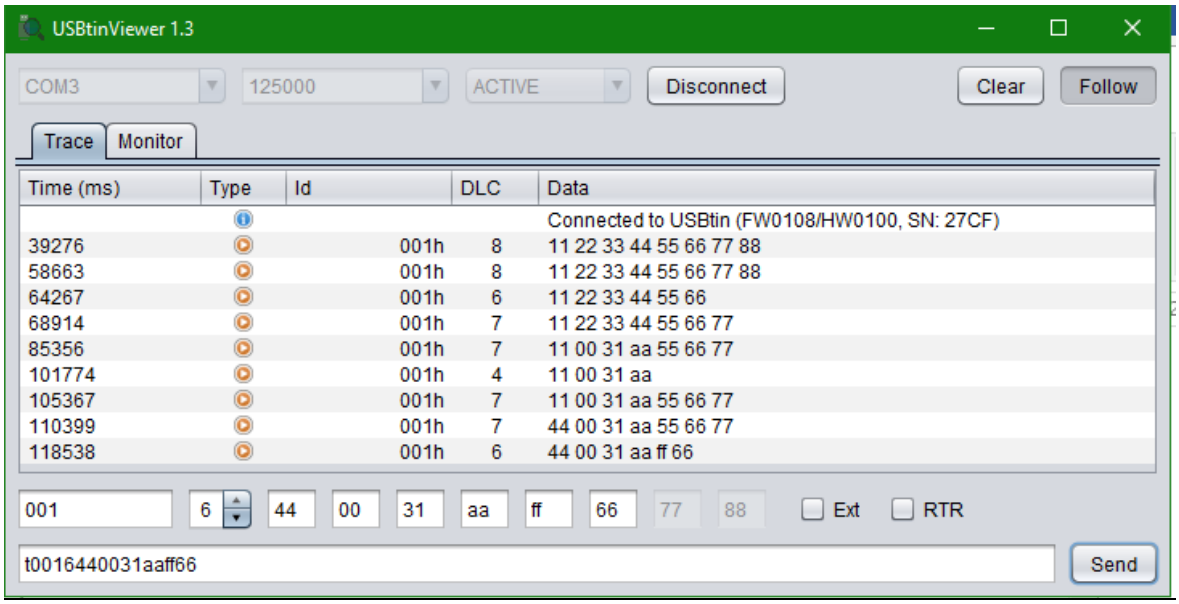


Fig. 5.2.1. Output window USBtinViwer, mensajes enviados. Fuente: propia.

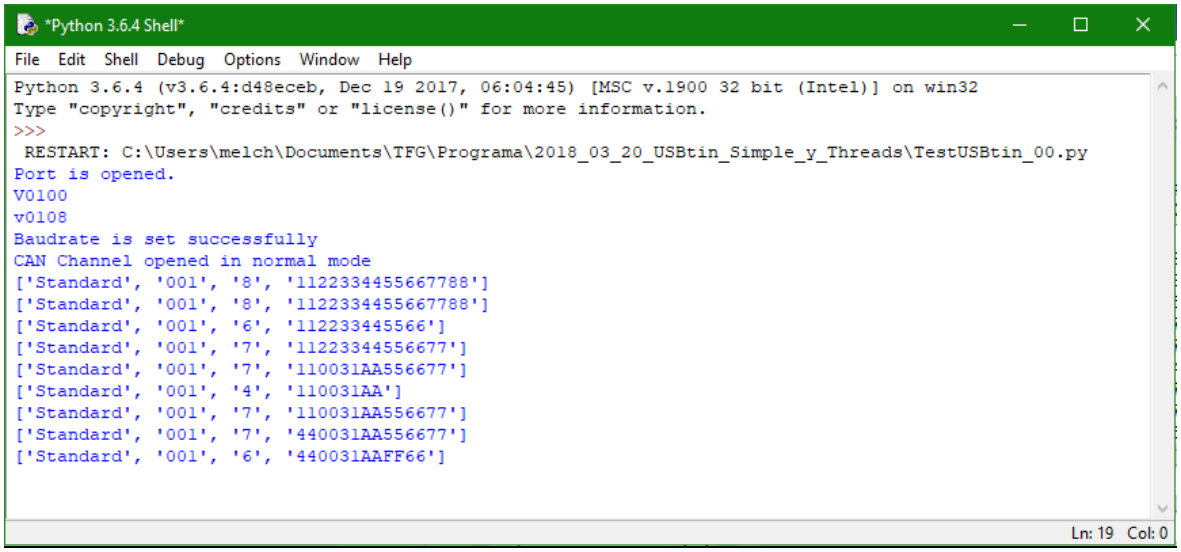


Fig. 5.2.2. Python Shell, mensajes recibidos. Fuente: propia.

Conclusiones

La puerta de enlace *GatewayUSBtin.py* es una herramienta muy útil para ver de forma más clara la información transmitida por un bus CAN. Su gran velocidad de procesamiento de datos hace que sea posible la lectura en tiempo real de lo que ocurre en un bus CAN.

Gracias a la programación de una pequeña interfaz gráfica, *GUI_GatewayUSBtin*, el funcionamiento de la puerta de enlace es muy sencillo y eficaz, dada su capacidad para poder conectarse o desconectarse y modificar los principales valores de conexión a un bus CAN. Estos valores son el puerto serie (COM), velocidad del bus (Baudrate), máscaras y filtros.

El software *opensource* BusMaster incrementa las posibilidades de trabajo del USBtin, haciéndolo mucho más completo. Se pueden crear bases de datos propias, de manera que el procesamiento de datos de los mensajes transmitidos por el bus CAN sea mucho más visual y comprensible.

Además, también puede trabajar con filtros y máscaras para poderse centrar en la señal de algún sensor en concreto conectado a un nodo del bus CAN. Un ejemplo de este caso sería, por ejemplo, en un camión que se sospecha que existe un problema de presión en algún circuito en el que hay un sensor conectado al bus CAN del vehículo. Usando como filtro el identificador de este sensor y activándolo con la máscara, la puerta de enlace únicamente dejará pasar mensajes de este sensor.

Finalmente, destacar que la fiabilidad presentada a lo largo de los test de verificación de funcionamiento es del 100%, lo que significa que la puerta de enlace es efectiva y está lista para usarse en cualquier tipo de proyecto llevado a cabo con dispositivos USBtin.

Agradecimientos

En primer lugar, a Manuel Moreno Eguílaz, por su ayuda tanto en la programación de la puerta de enlace como en la redacción del documento a lo largo de todo el proyecto.

A Thomas Fischl, creador del conversor USBtin, por la gran facilidad que ofrece dicho conversor para iniciarse en el mundo de transferencia de información mediante un bus CAN.

Por último, gracias a Guillermo Pérez, por la biblioteca usbTinLib, con la que se puede acceder desde Python al dispositivo USBtin de una forma muy sencilla.

Bibliografía

Referencias bibliográficas

[1] [http://www.deere.com/common/docs/html/services_and_support/onscreen_help/16-2/en/diagnostics_center/diagnostics_center_can_bus_info_canbus_info.htm]*. [www.deere.com, 15/05/2018]

[2] [https://www.fischl.de/usbtin/]*. [www.fischl.de, 15/05/2018]

[3] [https://www.kvaser.com/canking/]*. [www.kvaser.com, 22/05/2018]

[4] [https://www.kvaser.com/downloads-kvaser/?utm_source=software&utm_ean=7330130980860&utm_status=latest]*. [www.kvaser.com, 20/05/2018]

[5] [https://rbei-etas.github.io/busmaster/download.html]*. [rbei-etas.github.io,15/03/2018]

[6] [https://www.kvaser.com/]*. [www.kvaser.com, 15/05/2018]

[7] [https://www.kvaser.com/product/kvaser-leaf-light-hs/]*. [www.kvaser.com, 10/04/2018]

[8] [https://www.fischl.de/usbtin/]*. [www.fischl.de, 15/05/2018]

[9] FISCHL. [https://www.fischl.de/usbtin/]*. [www.fischl.de , 15/05/2018]

[10] [https://es.wikipedia.org/wiki/Bjarne_Stroustrup]*. [www.wikipedia.org, 20/05/2018]

[11] GUILLEM, P. *Control de múltiples adaptadores USBtin en Python,* TFG ETSEIB 2016. https://upcommons.upc.edu/handle/2117/99142, 25/05/2018.

[12] [https://www.kvaser.com/developer/canlib-sdk/]*.[www.kvaser.com, 24/05/2018]

[13] [https://polaridad.es/instalacion-herramientas-comunicaciones-serie-python/]*.[www.polaridad.es, 29/05/2018]

[14] [https://www.kvaser.com/downloads-kvaser/].*[www.kvaser.com, 27/05/2018]

[15] [https://www.tutorialspoint.com/python/tk_frame.htm]*.[www.tutorialspoint.com, 10/05/2018]